Ways to Improve Instantaneous Codes

Suppose we have an instantaneous code for symbols $a_1, \ldots, a_I$, with probabilities $p_1, \ldots, p_I$. Let l_i be the length of the codeword for a_i.

Under each of the following conditions, we can find a better instantaneous code, with smaller expected codeword length:

- If $p_1 < p_2$ and $l_1 < l_2$:

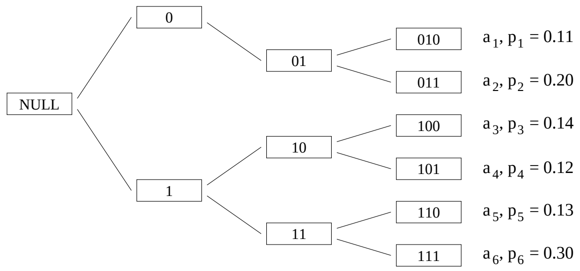
  Swap the codewords for a_1 and a_2.

- If there is a codeword of the form xby, where x and y are strings of zero or more bits, and b is a single bit, but there are no codewords of the form $xb'z$, where z is a string of zero or more bits, and $b' \neq b$.

  Change all the codewords of the form xby to xy. (Improves things if none of the p_i are zero, and never makes things worse.)

The Improvements in Terms of Trees

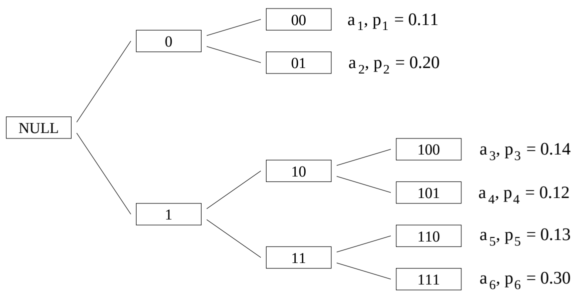
We can view these improvements in terms of the trees for the codes. Here's an example:



Two codewords have the form 01... but none have the form 00... (ie, there's only one branch out of the 0 node). We can therefore improve the code by deleting the surplus node.

Continuing to Improve the Example

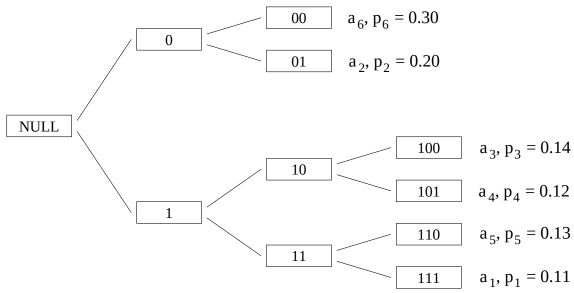
The result is the code shown below:



Now we note that a_6, with probability 0.30, has a longer codeword than a_1, which has probability 0.11. We can improve the code by swapping the codewords for these symbols.

The State After These Improvements

Here's the code after this improvement:



In general, after such improvements:

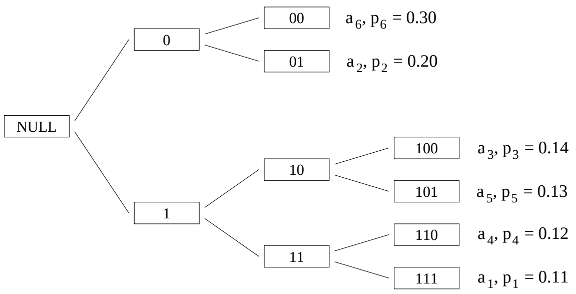
- The most improbable symbol will have a codeword of the longest length.
- There will be at least one other codeword of this length — otherwise the longest codeword would be a solitary branch.
- The second-most improbable symbol will also have a codeword of the longest length.

A Final Rearrangement

The codewords for the most improbable and second-most improbable symbols must have the same length. The most improbable symbol's codeword also has a "sibling" of the same length.

We can swap codewords to make this sibling be the codeword for the second-most improbable symbol.

For the example, the result is



Huffman Codes

We can use these insights about code trees to try to construct optimal codes.

We will prove later that the resulting *Huffman codes* are in fact optimal.

We'll concentrate on Huffman codes for a binary code alphabet. Non-binary Huffman codes are similar, but slightly messier.

Huffman Procedure for Binary Codes

Here's a recursive procedure to construct a Huffman code for a binary code alphabet:

procedure Huffman:

- **inputs:** symbols $a_1, \ldots, a_I$
  probabilities $p_1, \ldots, p_I$
- **output:** a code mapping $a_1, \ldots, a_I$ to codewords
- **if** $I = 2$:
  - Return the code $a_1 \mapsto 0$, $a_2 \mapsto 1$.
- **else**
  - Let $j_1, \ldots, j_I$ be a permutation of $1, \ldots, I$ for which $p_{j_1} \geq \cdots \geq p_{j_I}$.
  - Create a new symbol a', with associated probability $p' = p_{j_{I-1}} + p_{j_I}$.
  - Recursively call Huffman to find a code for $a_{j_1}, \ldots, a_{j_{I-2}}, a'$ with probabilities $p_{j_1}, \ldots, p_{j_{I-2}}, p'$. Let the codewords for $a_{j_1}, \ldots, a_{j_{I-2}}, a'$ in this code be $w_1, \ldots, w_{I-2}, w'$.
  - Return the code $a_{j_1} \mapsto w_1, \ \ldots, \ a_{j_{I-2}} \mapsto w_{I-2}, \ a_{j_{I-1}} \mapsto w'0, \ a_{j_I} \mapsto w'1$.

Proving that Binary Huffman Codes are Optimal

We can prove that the binary Huffman code procedure produces optimal codes by induction on the number of symbols, I.

For $I = 2$, the code produced is obviously optimal — you can't do better than using one bit to code each symbol.

For $I > 2$, we assume that the procedure produces optimal codes for any alphabet of size $I - 1$ (with any symbol probabilities), and then prove that it does so for alphabets of size I as well.

The Induction Step

Suppose the Huffman procedure produces optimal codes for alphabets of size $I-1$.

Let L be the expected codeword length of the code produced by the procedure when it is used to encode the symbols $a_1, \ldots, a_I$, having probabilities $p_1, \ldots, p_I$. Without loss of generality, let's assume that $p_i \geq p_{I-1} \geq p_I$ for all $i \in \{1, \ldots, I-2\}$.

The recursive call in the procedure will have produced a code for symbols $a_1, \ldots, a_{I-2}, a'$, having probabilities $p_1, \ldots, p_{I-2}, p'$, with $p' = p_{I-1} + p_I$. By the induction hypothesis, this code is optimal. Let its average length be L'.

The Induction Step (Continued)

Suppose some other instantaneous code for $a_1, \ldots, a_I$ had expected length less than L. We can modify this code so that the codewords for a_{I-1} and a_I are "siblings" (ie, they have the forms $x0$ and $x1$) while keeping its average length the same, or less.

Let the average length of this modified code be $\widehat{L}$, which must also be less than L.

From this modified code, we can produce another code for $a_1, \ldots, a_{I-2}, a'$. We keep the codewords for $a_1, \ldots, a_{I-2}$ the same, and encode a' as x. Let the average length of this code be $\widehat{L}'$.

The Induction Step (Conclusion)

We now have two codes for $a_1, \ldots, a_I$ and two for $a_1, \ldots, a_{I-2}, a'$. The average lengths of these codes satisfy the following equations:

$$\begin{aligned} L &= L' + p_{I-1} + p_I \\ \widehat{L} &= \widehat{L}' + p_{I-1} + p_I \end{aligned}$$

Why? The codes for $a_1, \ldots, a_I$ are like the codes for $a_1, \ldots, a_{I-2}, a'$, except that one symbol is replaced by two, whose codewords are one bit longer. This one additional bit is added with probability $p' = p_{I-1} + p_I$.

Since L' is the optimal average length, $L' \leq \widehat{L}'$. From these equations, we then see that $L \leq \widehat{L}$, which contradicts the supposition that $\widehat{L} < L$.

The Huffman procedure therefore produces optimal codes for alphabets of size I. By induction, this is true for all I.

What Have We Accomplished?

We seem to have solved the main practical problem: We now know how to construct an optimal code for any source.

But: This code is optimal **only** if the assumptions we made in formalizing the problem match the real situation.

Often they don't:

- Symbol probabilities may vary over time.
- Symbols may not be independent.
- There is usually no reason to require that $X_1, X_2, X_3, \ldots$ be encoded one symbol at a time, as $c(X_1)c(X_2)c(X_3)\cdots$.

We would require the last if we really needed instantaneous decoding, but usually we don't.

Example: Black-and-White Images

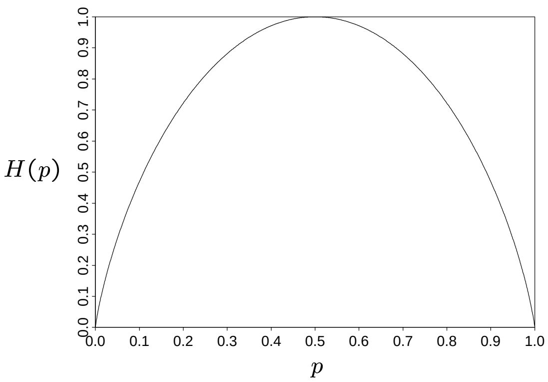
Recall the example from the first lecture, of black-and-white images. There are only two symbols — "white" and "black". The Huffman code is white $\mapsto$ 0, black $\mapsto$ 1.

This is just the obvious code. But we saw that various schemes such as run length coding can do better than this.

Partly, this is because the pixels are not independent. Even if they were independent, however, we would expect to be able to compress the image if black pixels are much less common than white pixels.

Entropy of a Binary Source

For a binary source, with symbol probabilities p and $1-p$, the entropy as a function of p looks like this:



$H(0.1) = 0.469$, so we hope to compress a binary source with symbol probabilities of 0.1 and 0.9 by more than a factor of two. We obviously can't do that if we encode symbols one at a time.

Solution: Coding Blocks of Symbols

We can do better by using Huffman codes to encode *blocks* of symbols.

Suppose our source probabilities are 0.7 for white and 0.3 for black. Assuming pixels are independent, the probabilities for blocks of two pixels will be

white white	$0.7 \times 0.7 = 0.49$
white black	$0.7 \times 0.3 = 0.21$
black white	$0.3 \times 0.7 = 0.21$
black black	$0.3 \times 0.3 = 0.09$

Here's a Huffman code for these blocks:

$WW \mapsto 0,\ WB \mapsto 10,\ BW \mapsto 110,\ BB \mapsto 111$

The average length for this code is 1.81, which is less than the two bits needed to encode a block the obvious way.